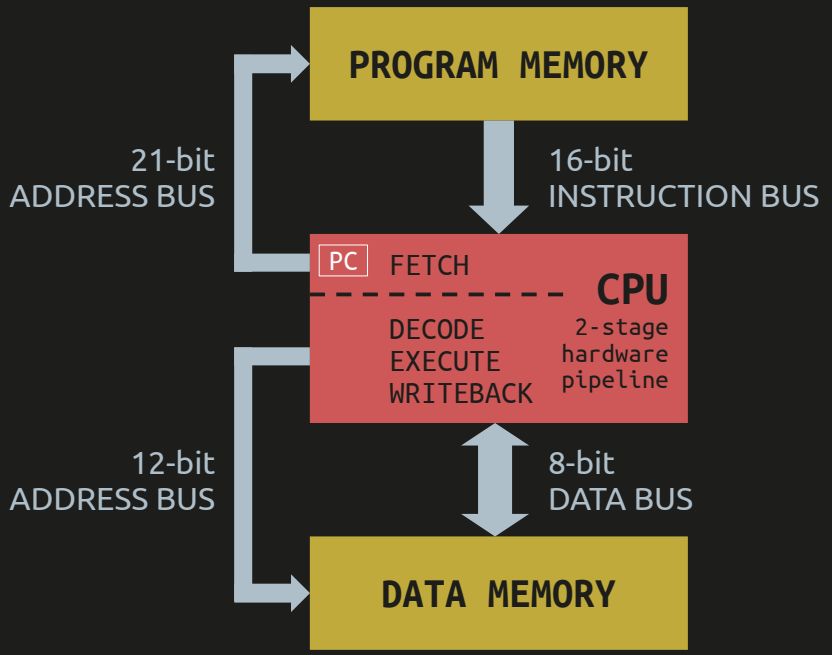
Chapitre 6

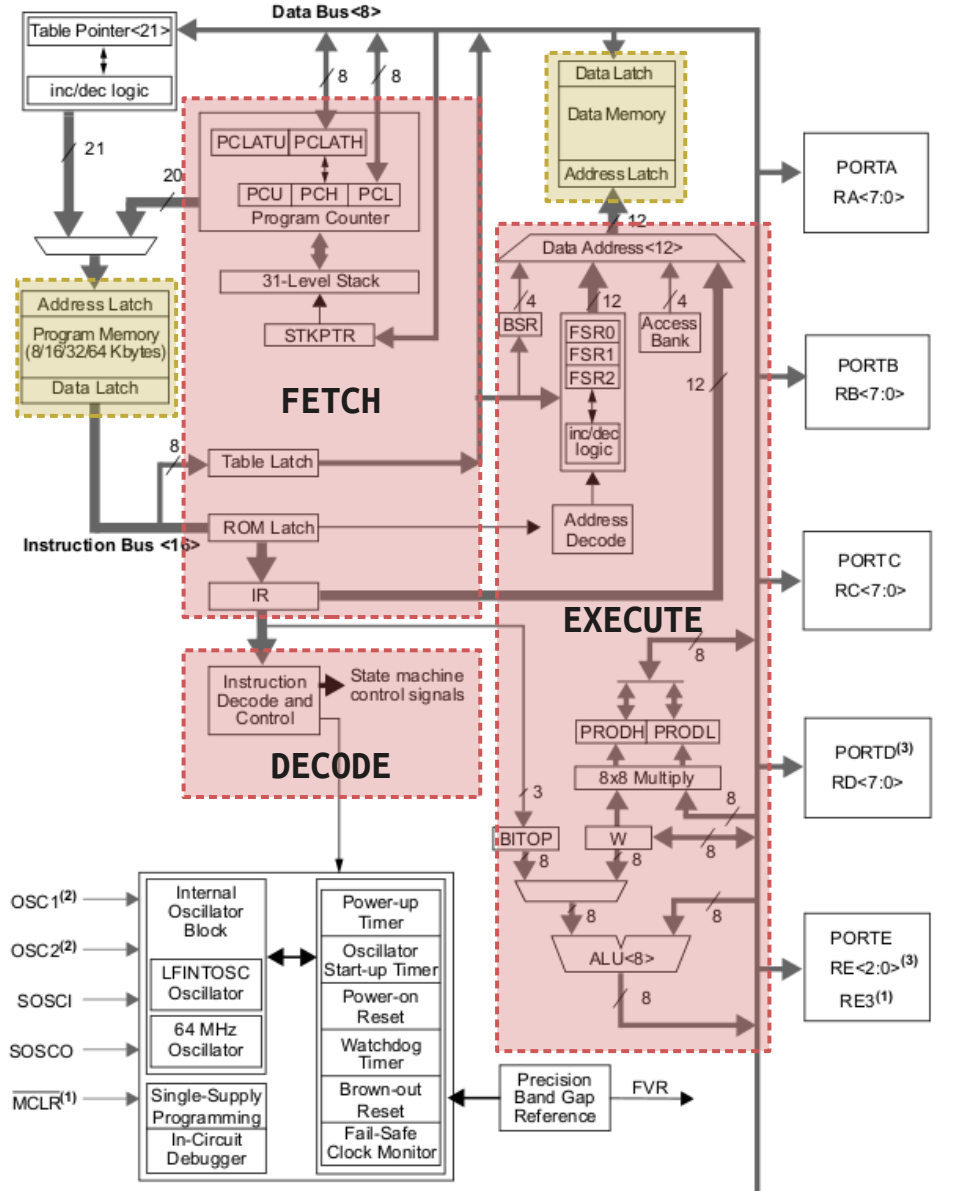
Assembleur PIC18

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

ARCHITECTURE PIC18





JEU D'INSTRUCTIONS PIC18

Instruction Set Architecture (ISA)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Liste des instructions

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb	LSb	Status Affected	Notes
BYTE-ORIENTED OPERATIONS							
ADDWF	f, d, a	Add WREG and f	1	0010 01da	ffff ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010 00da	ffff ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001 01da	ffff ffff	Z, N	1,2
CLRF	f, a	Clear f	1	0110 101a	ffff ffff	Z	2
COMF	f, d, a	Complement f	1	0001 11da	ffff ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110 001a	ffff ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110 010a	ffff ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110 000a	ffff ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000 01da	ffff ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010 11da	ffff ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100 11da	ffff ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010 10da	ffff ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011 11da	ffff ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100 10da	ffff ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001 00da	ffff ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101 00da	ffff ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination) 2nd word	2	1100 ffff 1111 ffff	ffff ffff ffff ffff	None	
MOVWF	f, a	Move WREG to f	1	0110 111a	ffff ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000 001a	ffff ffff	None	1, 2
NEGF	f, a	Negate f	1	0110 110a	ffff ffff	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	0011 01da	ffff ffff	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100 01da	ffff ffff	Z, N	
RRCF	f, d, a	Rotate Right f through Carry	1	0011 00da	ffff ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100 00da	ffff ffff	Z, N	
SETF	f, a	Set f	1	0110 100a	ffff ffff	None	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101 01da	ffff ffff	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	0101 11da	ffff ffff	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101 10da	ffff ffff	C, DC, Z, OV, N	
SWAPF	f, d, a	Swap nibbles in f	1	0011 10da	ffff ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110 011a	ffff ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001 10da	ffff ffff	Z, N	

Liste des instructions

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BIT-ORIENTED OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2
CONTROL OPERATIONS									
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ	n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None	
BRA	n	Branch Unconditionally	2	1101	0nnn	nnnn	nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None	
CALL	n, s	Call subroutine 1st word	2	1110	110s	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
CLRWDT	—	Clear Watchdog Timer	1	0000	0000	0000	0100	$\overline{TO}$, $\overline{PD}$	
DAW	—	Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	n	Go to address 1st word	2	1110	1111	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
NOP	—	No Operation	1	0000	0000	0000	0000	None	
NOP	—	No Operation	1	1111	xxxx	xxxx	xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None	
RESET		Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from Subroutine	2	0000	0000	0001	001s	None	
SLEEP	—	Go into Standby mode	1	0000	0000	0000	0011	$\overline{TO}$, $\overline{PD}$	

Liste des instructions

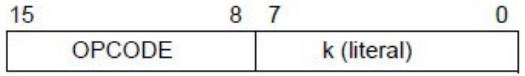
Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
LITERAL OPERATIONS									
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word	2	1110	1110	00ff	kkkk	None	
		to FSR(f) 1st word		1111	0000	kkkk	kkkk		
MOVLB	k	Move literal to BSR<3:0>	1	0000	0001	0000	kkkk	None	
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS									
TBLRD*		Table Read	2	0000	0000	0000	1000	None	
TBLRD*+		Table Read with post-increment		0000	0000	0000	1001	None	
TBLRD*-		Table Read with post-decrement		0000	0000	0000	1010	None	
TBLRD+*		Table Read with pre-increment		0000	0000	0000	1011	None	
TBLWT*		Table Write	2	0000	0000	0000	1100	None	
TBLWT*+		Table Write with post-increment		0000	0000	0000	1101	None	
TBLWT*-		Table Write with post-decrement		0000	0000	0000	1110	None	
TBLWT+*		Table Write with pre-increment		0000	0000	0000	1111	None	

77-instruction ISA (Instruction Set Architecture)

JEU D'INSTRUCTIONS PIC18

Format des instructions

Literal operations

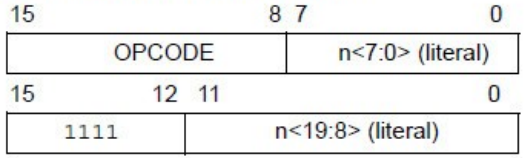


k = 8-bit immediate value

MOVLW 7Fh

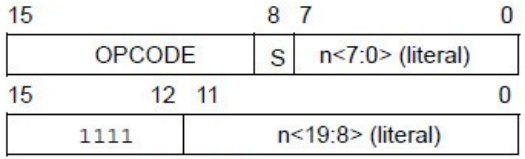
Control operations

CALL, GOTO and Branch operations



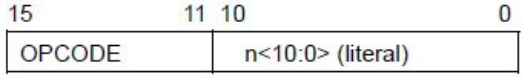
n = 20-bit immediate value

GOTO Label

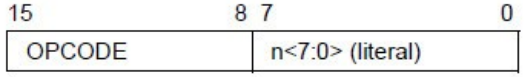


S = Fast bit

CALL MYFUNC



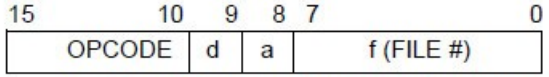
BRA MYFUNC



BC MYFUNC

Byte-oriented file register operations

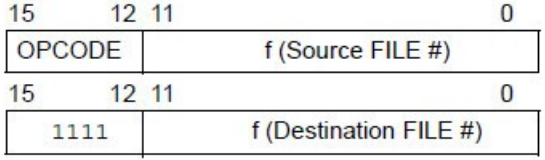
Example Instruction



d = 0 for result destination to be WREG register
d = 1 for result destination to be file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

ADDWF MYREG, W, B

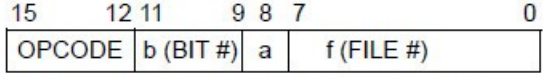
Byte to Byte move operations (2-word)



f = 12-bit file register address

MOVFF MYREG1, MYREG2

Bit-oriented file register operations



b = 3-bit position of bit in file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

BSF MYREG, bit, B

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

DU C À L'ASSEMBLEUR

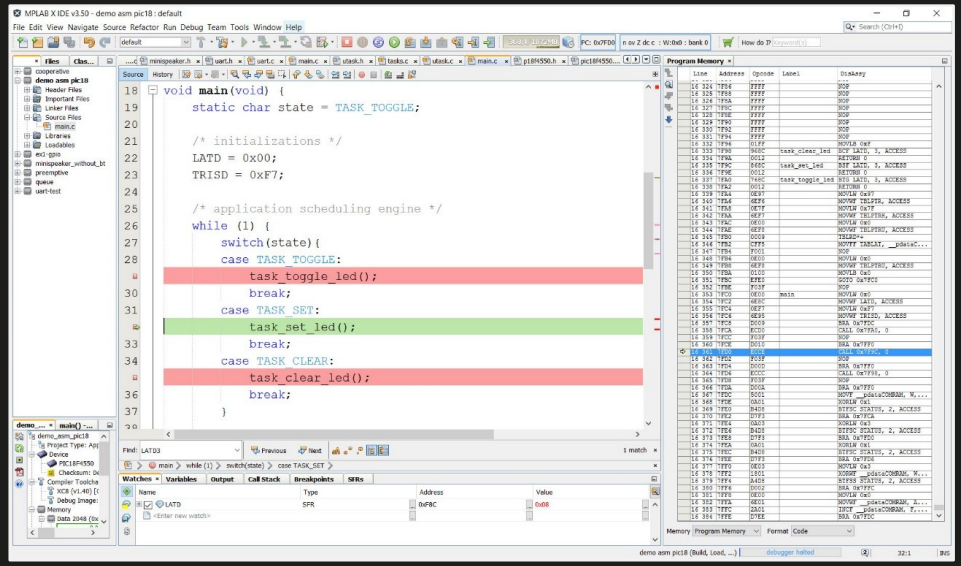
ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Afin de mieux comprendre le jeu d'instructions du PIC18, nous traduirons pas un pas un programme en C vers de l'assembleur PIC18.

Ce travail peut être refait et adapté sur vos propres machines en installant l'IDE MPLAB X et la toolchain XC8 , puis en utilisant le mode mode.

Vous trouverez les outils sur Moodle et les conseils d'installation sur les supports de TP.



Programme C à traduire

```c
/* CPU specific features configuration */
#pragma config FEXTOSC = OFF CLKOUTEN = OFF
#pragma config RSTOSC = HFINTOSC_64MHZ
#pragma config MCLRE = EXTMCLR PWRTE = OFF
#pragma config BOREN = SBORDIS DEBUG = OFF

#include <pic18f27k40.h>

#define TASK_TOGGLE  1
#define TASK_SET     2
#define TASK_CLEAR   3

void toggle_led_D2(void) {
    #asm
        BTG  LATA, 4
    #endasm
}

void set_led_D2(void) {
    LATA |= 0x10;
}

void clear_led_D2(void) {
    LATAbits.LATA4 = 0;
}
```

```c
/* main entry point - dummy application */
void main(void) {
    static char state;

    /* initializations */
    state = TASK_TOGGLE;
    LATA  = 0x00;
    TRISA = 0x7F;

     /* scheduling engine */
    while (1) {
        switch(state) {
            case TASK_TOGGLE:
                toggle_led_D2();
                break;
            case TASK_SET:
                set_led_D2();
                break;
            case TASK_CLEAR:
                clear_led_D2();
                break;
            }

        /* application state update */
        if (state == TASK_CLEAR)
            state = 0;

        state++;
    }
}
```

Insertion d'assembleur dans le C

Les programmes sont majoritairement écrit en langage C. Ceci facilite la lisibilité et la maintenance du code.

Parfois de courtes portions de codes seront écrites en assembleur. Ce sera le cas pour des fonctions spécifiques, avec l'objectif d'optimiser le programme (gain de mémoire programme, d'empreinte mémoire ou accélération).

Full C function

```
void toggle_led_D2(void) {

    LATAbits.LATA4 ^= 1;

}
```

C function with intermixed assembly

```
void toggle_led_D2(void) {
    #asm
        BTG  LATA, 4
    #endasm
}
```

Allocation statique

Pour faciliter l'exercice, les variables seront toutes déclarées en **static**. L'adresse de la variable sera générée pendant le processus de compilation (à l'édition des liens) et son adresse sera connue pendant toute la durée d'exécution du programme.

Nous n'utiliserons pas la pile (*stack*) et aucune variable non-static ne sera utilisée ici.

C language

```c
/* Static variable declaration */
static char state;
```

XC8 assembly language

```
; Reserve 1 byte in Access Bank
PSECT <static_section_name>, class=BANK0,space=1
state: ds 1
```

Mode d'adressage immédiat

Le mode d'adressage immédiat correspond à la manipulation d'une constante.

Autrement dit, le code binaire de l'opération contient immédiatement la valeur binaire de la constante.

Sur des architectures RISC 32 bits, les constantes sont généralement limitées à 16 bits. Pour la gamme PIC18, celles-ci sont limitées à 8 bits.

Les instructions PIC18 utilisant l'adressage immédiat sont appelées "*literal operation*".

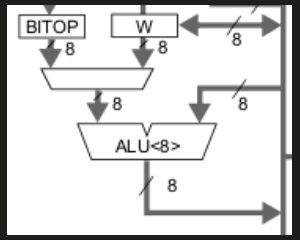
C language

```
/* initializations */
state = TASK_TOGGLE;
LATA  = 0x00;
TRISA = 0x7F;
```

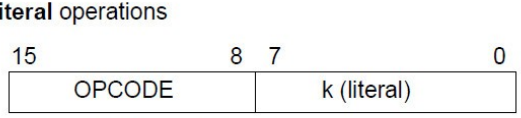
PIC18 assembly language

Instructions utilisant le mode d'adressage immédiat

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word MSb LSb	Status Affected	Notes
LITERAL OPERATIONS					
ADDLW k	Add literal and WREG	1	0000 1111 kkkk kkkk	C, DC, Z, OV, N	
ANDLW k	AND literal with WREG	1	0000 1011 kkkk kkkk	Z, N	
IORLW k	Inclusive OR literal with WREG	1	0000 1001 kkkk kkkk	Z, N	
LFSR f, k	Move literal (12-bit) 2nd word to FSR(f) 1st word	2	1110 1110 00ff kkkk 1111 0000 kkkk kkkk	None	
MOVLB k	Move literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLW k	Move literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW k	Multiply literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW k	Subtract WREG from literal	1	0000 1000 kkkk kkkk	C, DC, Z, OV, N	
XORLW k	Exclusive OR literal with WREG	1	0000 1010 kkkk kkkk	Z, N	



Literal operations



k = 8-bit immediate value

← Format des instructions utilisant une valeur immédiate 8 bits
(certaines utilisent des valeurs sur 12 bits)

Utilisation de la RAM

La RAM est constituée de 16 banques de 256 octets (16 x 256 = 4 kB max).

À un instant donné, seules deux banques sont accessibles : celle explicitement désignée par le BSR (*Bank Select Register*) et l'*access bank* (top half of bank #0 + SFR bank).

Les instructions contiennent un bit <a> dans leur code binaire opératoire

<a> = 0 indique la sélection de l'*access bank* ()

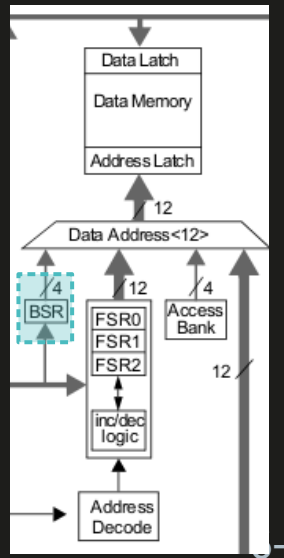
<a> = 1 indique la sélection par valeur du BSR.

Two ways of accessing the same data in bank #0

```
ADDWF    <8bit_relative_address>, 0, 0    ; <a>=0 → access bank selected
                                          ; bank0   0x000-0x05F (GPR)
                                          ; bank15  0xF60-0xFFF (SFR)
```

Or

```
MOVLB     0x0                             ; Move literal (0x0) to BSR → select bank #0
ADDWF     <8bit_relative_address>, 0, 1   ; <a>=1 → bank selected with BSR
                                          ; bank0   0x000-0x0FF (GPR)
```



Toutes les instructions de contrôle en C (`if`, `else if`, `else`, `switch/case`, `while`, `for`, ...) réalisent un saut d'une instruction à une autre.

Le *Program Counter* (PC) voit alors sa valeur écrasée pour correspondre à l'adresse de l'instruction à atteindre.

C language

```c
/* main entry point */
void main(void) {

     /* User code part 1 */


     /* main routine */
     while (1) {
          /* User code part 2 */

     }
}
```

PIC18 assembly language

Saut inconditionnel

Deux instructions permettent de faire un **saut inconditionnel** : `GOTO` et `BRA` (*branch*).

`GOTO` utilise une adresse absolue et peut donc accéder à toute la mémoire programme.

`BRA` utilise une adresse relative au PC et ne peut accéder qu'à une petite partie de la mémoire programme.

En assembleur, on peut donner un **label** (ou plusieurs) à une instruction ou une donnée, pour y avoir accès sans pour autant connaître son adresse réelle.

Absolute addressing

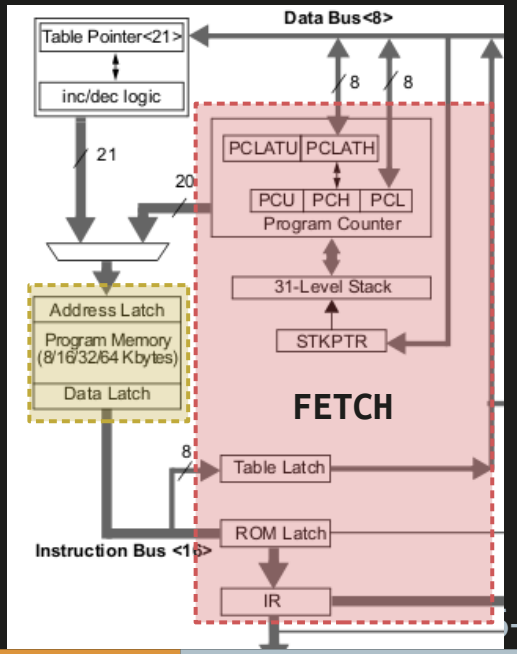
```
main:
     ; User code part 1

main_loop1:
     ; User code part 2
     GOTO    main_loop1
```

Relative addressing

```
main:
     ; User code part 1

main_loop1:
     ; User code part 2
     BRA     main_loop1
```



La mémoire programme est adressable sur 21 bits (max 2 MB ou 1 Mword).

Pour accéder tout cet espace, l'opérande de l'instruction `GOTO` doit aussi être sur 21 bits.

En réalité l'opérande est uniquement 20 bits car chaque instruction est stockée sur deux octets, le bit de poids faible du PC n'a pas d'impact sur l'instruction désignée.

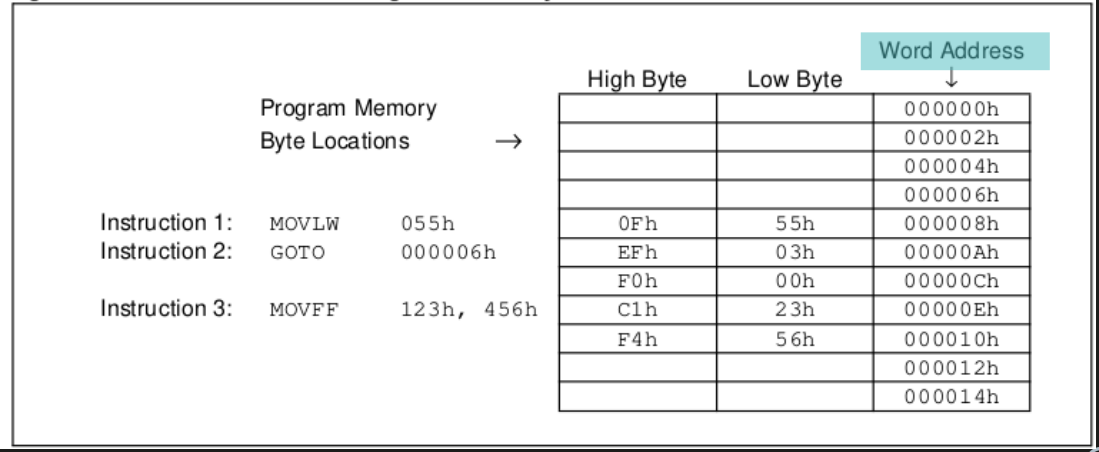
Pour contenir ces 20 bits, l'instruction `GOTO` est codée sur 4 octets (2 mots mémoire).

GOTO — **Unconditional Branch**

Syntax:	[*label*] GOTO k
Operands:	$0 \le k \le 1048575$
Operation:	$k \rightarrow PC<20:1>$ $0 \rightarrow PC<0>$,
Status Affected:	None
Encoding:	

1st word (k<7:0>)	1110	1111	k_7kkk	$kkkk_0$
2nd word (k<19:8>)	1111	$k_{19}kkk$	kkkk	$kkkk_8$

Figure 7-2: Instructions in Program Memory



Saut inconditionnel

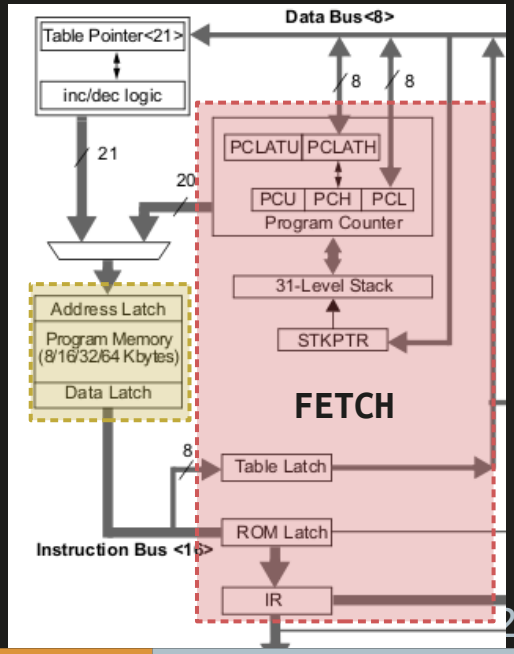
L'instruction BRA utilise comme opérande une adresse relative à PC.

L'offset est codé sur 11 bits, correspondant à un offset de -1024 à +1023 octets (et non des mots). L'instruction est donc codée sur 2 octets et donne une empreinte mémoire plus faible que le GOTO, mais ne peut pas accéder à toute la mémoire programme.

Pour autant que BRA n'est pas plus rapide que GOTO (purge du pipeline).

Mnemonic, Operands		Description		Cycles	16-Bit Instruction Word MSb			LSb
CONTROL OPERATIONS								
BRA	n	Branch Unconditionally		2	1101	0nnn	nnnn	nnnn
GOTO	n	Go to address	1st word	2	1110	1111	kkkk	kkkk
			2nd word		1111	kkkk	kkkk	kkkk

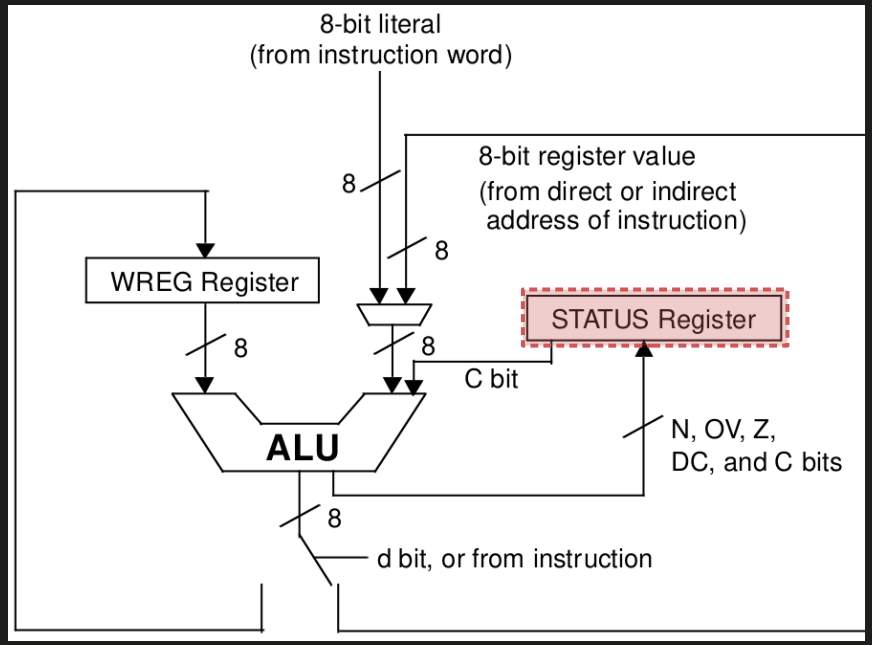
2 cycles car on devra vider le pipeline, le CPU ayant déjà récupéré l'instruction qui suit le BRA.



Saut conditionnel

Après chaque opération*, l'ALU met à jour les ***flags*** du ***STATUS register***.

*Voir la colonne « Status Affected » de la liste d'instructions pour savoir quels *flags* sont mis à jour par les instructions.



Register 5-1: STATUS Register

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative, (ALU MSb = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude, which causes the sign bit (bit7) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW, and SUBWF instructions
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the 2's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the bit4 or bit3 of the source register.

bit 0 **C:** Carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW, and SUBWF instructions
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the 2's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

Pour réaliser un test en assembleur, nous utilisons les **sauts conditionnels**.

Un saut conditionnel sera réalisé (ou non) en fonction de la valeur des *flags* du STATUS register, valeurs qui ont été mises à jour par la précédente opération.

Si la condition est validée, le saut sera effectué.

Sinon, l'instruction de saut n'est pas réalisée et on passe à l'instruction suivante.

C language

```c
/* application state update */
if ( state == TASK_CLEAR )
     state = 0;

state++;
```

PIC18 assembly language

Il est maintenant temps de traduire le séquenceur en assembleur !

```
 /* scheduling engine */
while (1) {
     switch(state) {
          case TASK_TOGGLE:
               toggle_led_D2();
               break;
          case TASK_SET:
               set_led_D2();
               break;
          case TASK_CLEAR:
               clear_led_D2();
               break;
          }

     /* application state update */
     if (state == TASK_CLEAR)
          state = 0;

     state++;
}
```

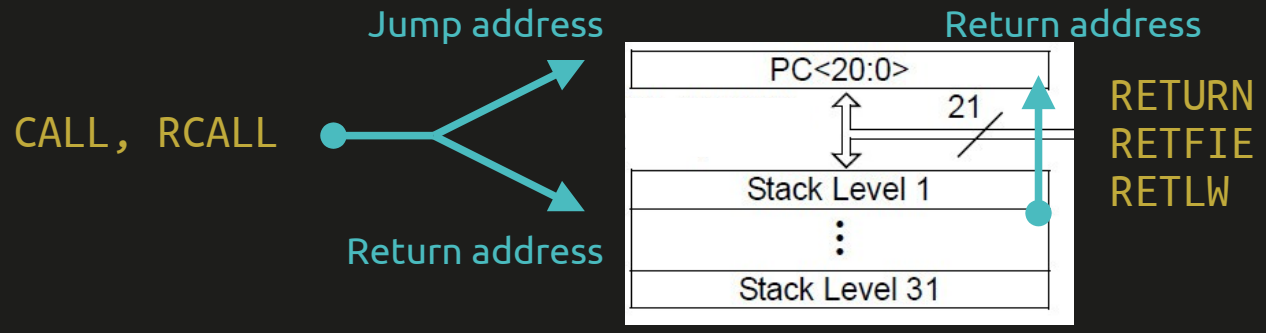
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution (pas unique !)

L'instruction `CALL` lance l'appel de fonction à partir d'une adresse absolue (instruction sur 4 octets) tandis que `RCALL` utilise une adresse relative (-1024/+1023, sur 2 octets).

Ces deux instructions sauvegardent la valeur courante du *Program Counter* en sommet de pile matérielle, avant de modifier PC pour accéder à la fonction.

L'instruction `RETURN` retire cette adresse de retour de la *stack* et la ré-écrit dans le *Program Counter* de sorte à revenir à l'instruction qui suit l'appel de fonction.



31-stage hardware stack

Appel de fonction : sauvegarde du contexte

CALL — Call Subroutine

Syntax: [*label*] CALL k, s

Operands: $0 \le k \le 1048575$, $s \in [0,1]$

Operation:
(PC)+ 4 → TOS,
k → PC<20:1>,
0 → PC<0>,
if s = 1
(WREG) → WREGS,
(STATUS) → STATUSS,
(BSR) → BSRS

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	110s	$k_7$kkk	kkkk_0
2nd word (k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: Subroutine call of entire 2M byte memory range. First, return address (PC+ 4) is pushed onto the return stack (20-bits wide).

If 's' = 1, the WREG, STATUS and BSR Registers are also pushed into their respective Shadow Registers, WREGS, STATUSS and BSRS.

If 's' = 0, no update occurs.

Then the 20-bit value 'k' is loaded into PC<20:1>. CALL is a two-cycle instruction.

Words: 2

Cycles: 2

Pourquoi max 1048575 ?

TOS = Top Of Stack

Pourquoi (PC)+4 → TOS ?

Pourquoi k → PC<20:1> ?

Sauvegarde matérielle (*shadow registers*)

's' = shadow registers bit

Les trois registre WREG, STATUS et BSR sont liés au contexte de la fonction en cours. Lorsqu'une autre fonction est appelée, il faut sauvegarder le contexte de la fonction appelante.

Le PIC18 propose une sauvegarde matérielle de ces trois registres dans des *shadow registers* (registres inaccessibles par le développeur).

High-prio IRQ : sauvegarde matérielle (*shadow registers*)

Low-prio : sauvegarde logicielle (RAM) par le développeur

Opérations sur les octets

Comme tout MCU, le PIC18 possède des instructions opérant sur les octets.

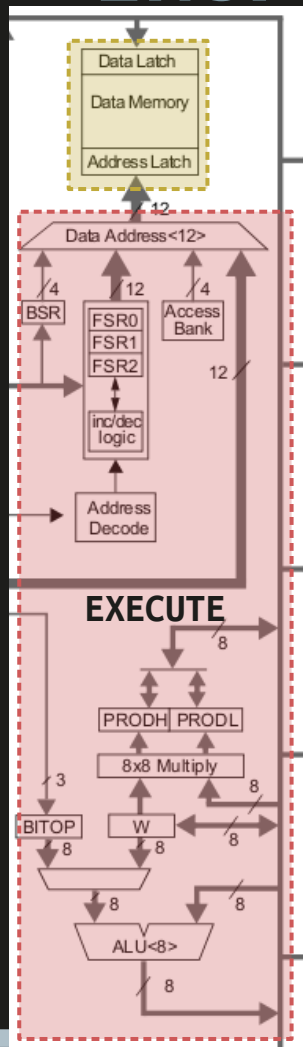
Cela regroupe les opérations de l'ALU, du multiplieur mais aussi celles de manipulation des données (opérations load/store) telles que :

- du CPU vers la mémoire (`MOVWF`, 2 bytes, 1 cycle)
- de la mémoire vers le CPU (`MOVF` 2 bytes, 1 cycle)
- de la mémoire vers la mémoire (`MOVFF`, 4 bytes, 2 cycles)

C language

```c
/* activate LED state */
void set_led(void) {
     LATA |= 0x10;
}
```

PIC18 assembly language



Opérations sur les bits

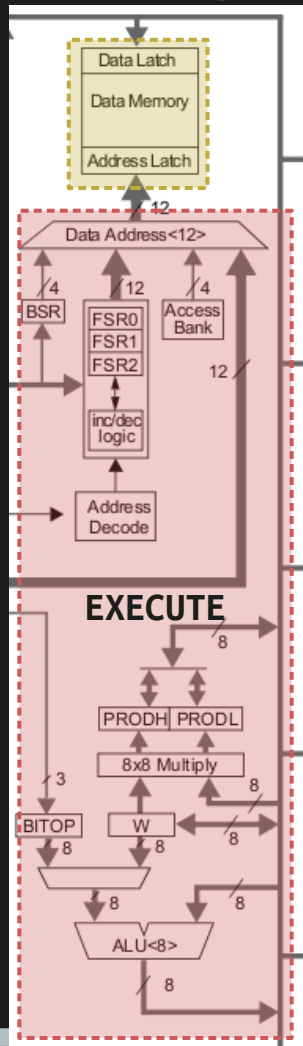
Microchip a également développé des instructions opérant sur les bits, fait peu courant sur des architectures modernes (contrairement aux instructions orientées octets).

Avec le PIC18, il est alors possible de lire, écrire ou tester un bit précis d'un registre. Ceci apporte un certain avantage et rend la configuration de périphérique plus aisée.

C language

```c
/* deactivate LED state */
void clear_led(void) {
     LATAbits.LATA4 = 0;
}
```

PIC18 assembly language

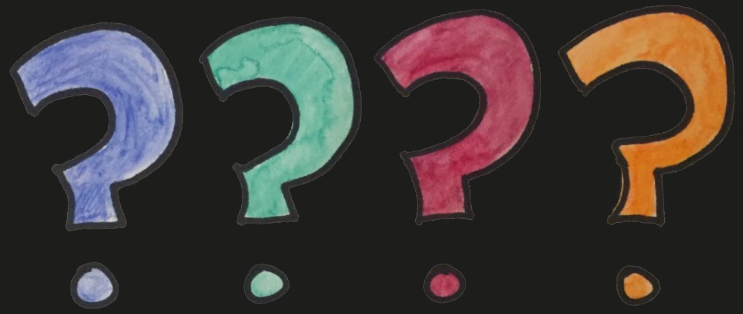


Traduction C → asm complète

À vous de jouer !

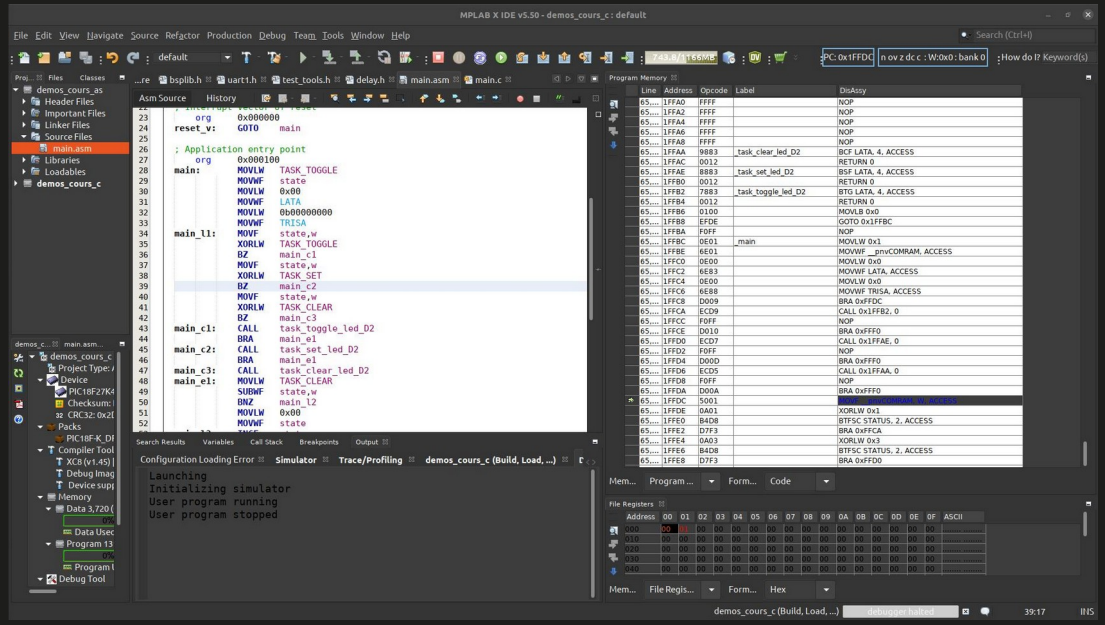
Traduisez le code C complet en assembleur.

Tout à déjà été vu dans les diapos précédentes !

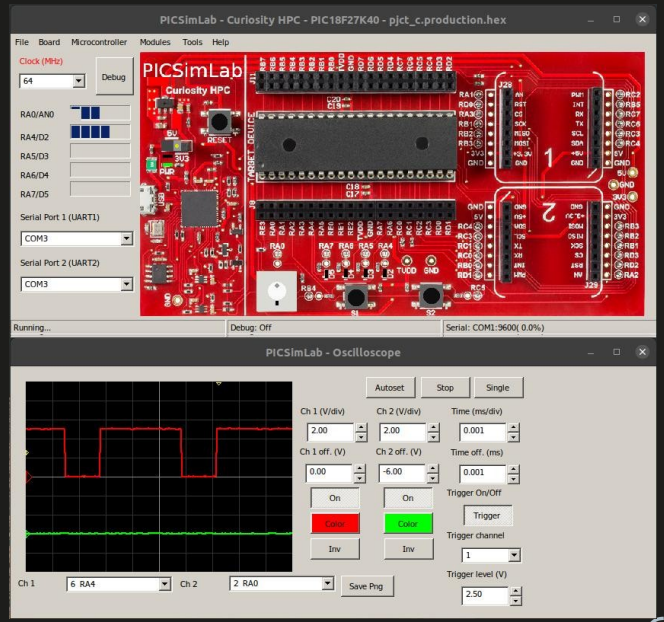


Dans le répertoire `tp/disco/apps/demos_cours`, il vous est proposé des projets pré-créés sous XC8 v1.45 (programme C et ASM PIC18). Ces projets vous permettront de pouvoir retravailler et durcir votre compréhension des processeurs PIC18 et de l'enseignement

MPLABX Simulator with debugger



PICSimLab



Microchip propose l'accès gratuit à ses outils de développement, notamment ses chaînes de compilation en version LITE. Ces versions ne permettent pas de lever toutes les options d'optimisation à la compilation.

Sous XC8 et C18, les versions payantes permettent notamment d'offrir au compilateur C l'accès aux instructions suivantes.

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word MSb	LSb	Status Affected
ADDFSR f, k	Add literal to FSR	1	1110 1000	ffkk kkkk	None
ADDULNK k	Add literal to FSR2 and return	2	1110 1000	11kk kkkk	None
CALLW	Call subroutine using WREG	2	0000 0000	0001 0100	None
MOVSF z_s, f_d	Move z_s (source) to 1st word f_d (destination) 2nd word	2	1110 1011 1111 ffff	0zzz zzzz ffff ffff	None
MOVSS z_s, z_d	Move z_s (source) to 1st word z_d (destination) 2nd word	2	1110 1011 1111 xxxx	1zzz zzzz xzzz zzzz	None
PUSHL k	Store literal at FSR2, decrement FSR2	1	1110 1010	kkkk kkkk	None
SUBFSR f, k	Subtract literal from FSR	1	1110 1001	ffkk kkkk	None
SUBULNK k	Subtract literal from FSR2 and return	2	1110 1001	11kk kkkk	None

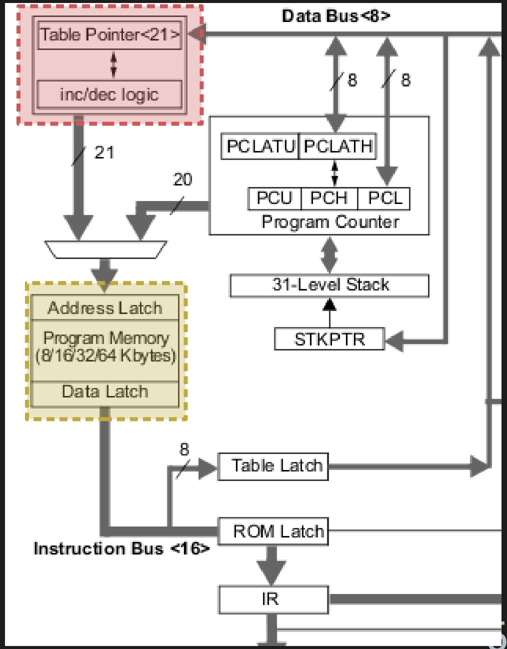
Placer des données en mémoire programme

Les PIC18 offrent une architecture de Harvard et par défaut une faible empreinte de mémoire donnée (application de contrôle). Il est néanmoins possible de manipuler des données chargées en mémoire programme, transformant ainsi l'architecture en processeur de Von Neumann (solution lente).

En langage C, on utilise les classes de stockage `rom` ou `ram` (par défaut) afin de forcer les outils de compilation à utiliser les instructions associées.

Par exemple : `rom char foo` ou `ram char foo/char foo`.

DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS								
TBLRD*	Table Read	2	0000	0000	0000	1000	None	
TBLRD*+	Table Read with post-increment		0000	0000	0000	1001	None	
TBLRD*-	Table Read with post-decrement		0000	0000	0000	1010	None	
TBLRD+*	Table Read with pre-increment		0000	0000	0000	1011	None	
TBLWT*	Table Write	2	0000	0000	0000	1100	None	
TBLWT*+	Table Write with post-increment		0000	0000	0000	1101	None	
TBLWT*-	Table Write with post-decrement		0000	0000	0000	1110	None	
TBLWT+*	Table Write with pre-increment		0000	0000	0000	1111	None	



ALLER PLUS LOIN

Document de référence :

PICmicro 18C MCU Family Reference Manual
(voir archive de TP)

PIC18(L)F27/47K40 datasheet
(voir archive de TP)

CONTACT

Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)